Scalable Resource Management in High Performance Computers *

Eitan Frachtenberg, Fabrizio Petrini, Juan Fernandez, and Salvador Coll
CCS-3 Modeling, Algorithms, and Informatics Group
Computer and Computational Sciences (CCS) Division
Los Alamos National Laboratory (LANL)
{eitanf,fabrizio,juanf,scoll}@lanl.gov

Abstract

Clusters of workstations have emerged as an important platform for building cost-effective, scalable, and highly-available computers. Although many hardware solutions are available today, the largest challenge in making large-scale clusters usable lies in the system software. In this paper we present STORM, a resource management tool designed to provide scalability, low overhead, and the flexibility necessary to efficiently support and analyze a wide range of job-scheduling algorithms. STORM achieves these feats by using a small set of primitive mechanisms that are common in modern high-performance interconnects. The architecture of STORM is based on three main technical innovations. First, a part of the scheduler runs in the thread processor located on the network interface. Second, we use hardware collectives that are highly scalable both for implementing control heartbeats and to distribute the binary of a parallel job in near-constant time. Third, we use an I/O bypass protocol that allows fast data movements from the file system to the communication buffers in the network interface and vice versa.

The experimental results show that STORM can launch a job with a binary of $12\,MB$ *on a* 64*-processor,* 32*-node cluster in less than* $250\,ms$*. This paper provides experimental and analytical evidence that these results scale to a much larger number of nodes. To the best of our knowledge, STORM significantly outperforms existing production schedulers in launching jobs, performing resource management tasks, and gang-scheduling tasks.*

Keywords*: Cluster Computing, Resource Management, Job Scheduling, Gang Scheduling, Parallel Architectures, Quadrics Interconnect, I/O bypass*

1. Introduction

Recent improvements in commodity processors and networks, combined with their attractive price/performance ratio, have made clusters of workstations a popular form of high performance computing. For example, RLX[1], Compaq[2] and HP[3] have recently introduced high-density *blade servers* that incorporate low-power processors. Several hundreds of these processors can be integrated in a single rack, and it is foreseeable that in the near future clusters with thousands of processors will quickly move out of the boundaries of research labs and academic research and will become widespread in the commercial world.

Although powerful hardware solutions are already available, the largest challenge to make these clusters usable lies in the system software. The scalability of resource management, job scheduling, and job launching are important aspects that are often overlooked.

Many run-time environments use a globally mounted file system, such as NFS, when they have to move executables, for example, when they spawn the processes of a job.[4] This design, where potentially many clients are accessing a single file on a single server at the same time is inherently non-scalable. In such environments, the typical method of launching a job is a shell script that loops over remote-shell commands (that use TCP/IP), to start processes on remote nodes. Although this is not a problem on small-scale clusters, this approach can have severe performance and scalability limitations on larger systems with several hundreds (or possibly thousands) of nodes.

The ParPar [18] cluster environment addresses the problem of distributing control messages from a management node to a set of clients by implementing a special-purpose multicast protocol. This protocol, called Reliable DataGram Multicast (RDGM), broadcasts UDP datagrams on the network and adds selective multicast and reliability. Each datagram is prepended by a bit string that identifies the set of destinations, and each node in the destination set sends an

[1] http://www.rlxtechnologies.com

[2] http://www.compaq.com/products/servers/proliant-bl/e-class/index.html

[3] http://www.hp.com/products1/servers/blades

[4] http://www.openpbs.org

acknowledgment to the management node after the successful delivery of the broadcast datagram. By using RDGM, a job can be launched in a few tens of seconds on a cluster with 16 nodes, with relatively good scalability.

GLUnix is an operating system middleware for clusters, designed to provide transparent remote execution, load balancing, coscheduling of parallel jobs and fault detection. In [15] the authors note that the overhead in the master node, when forking a parallel job, increases by a small amount (an average of $220\ \mu s$ per node). Also, one-to-many communication patterns scale relatively well, at only $230\ \mu s$ per node. When GLUnix launches a job, remote execution messages are sent from the management node to all the daemons that will run the job. Each of these daemons generate a reply message, indicating success or failure. When performing remote execution to many nodes (more than 32, in the experimental results shown in [15]) the replies from earlier daemons in the communication schedule collide with the remote execution requests sent to later daemons on the switched Ethernet, causing a substantial performance degradation. Thus, many-to-one communication patterns using TCP/IP over Ethernet may exhibit poor scalability.

Scalability problems are already evident in ASCI-scale machines, with thousands of nodes. The Computational Plant (Cplant) at Sandia National Laboratories includes several large-scale parallel computers composed of commodity computing and networking components. In order to enhance scalability, Cplant uses a high-performance interconnect, Myrinet [5], and a custom lightweight communication protocol based on Portals [6]. When the run-time environment of Cplant launches a job, it first identifies a group of active worker nodes, organizes them in a logical tree structure and then fans out the executable.

Many recent research results show that job scheduling algorithms can substantially improve scalability, responsiveness, resource utilization and usability of a large-scale parallel machine [2] [12]. Unfortunately, the body of work developed in the last few years has not yet led to any practical applications/implementations of such gang scheduling and coscheduling algorithms in parallel clusters. We argue that one of the main problems is the lack of flexible and efficient run-time systems that can support the implementation and evaluation of new scheduling algorithms, which are expected to replace conventional, space-shared, schedulers.

In this paper we present STORM (Scalable TOol for Resource Management). STORM provides a number of technical results that can pave the way to research advances in the area of resource management and job scheduling. STORM achieves these by using a software architecture that can exploit low-level network features for resource-management tasks. Although STORM can be implemented over any modern interconnect, we used the Quadrics network (QsNET) [22] because its easy programmability and superb performance. Large-scale clusters based on the QsNET are already installed at CEA (France), LLNL, ORNL, PSC (largest unclassified computer in the world), and LANL.

At the core of STORM there are three main technical innovations. First, STORM's management daemons can exchange control messages with a simple send/receive mechanism that uses the lowest-level interface of underlying interconnect. In the case of QsNET, these can be executed by threads that run in the thread processor of the network interface. These threads can process incoming messages and perform protocol processing without interrupting the computing node. Second, STORM fully exploits hardware support for broadcast for scalable dissemination of control and synchronization messages (QsNET can also combine the acknowledgments in the network switches). Third, the processors in the network interface can be used for an extremely lightweight I/O by-pass protocol, that allows interactions with the file system with almost no measurable overhead on the processing nodes. These innovations, combined with lightweight user-level daemons, translate to implementations of resource-management primitives that are low latency and scalable. Another key feature of STORM is it flexibility, designed to allow quick implementation and testing of new scheduling algorithms. This paper also analyzes the performance of gang-scheduling in STORM.

The rest of this paper is organized as follows. Section 2 describes the architecture of STORM, the main design goals, and the interconnect features that are used by STORM. In Section 3 we evaluate the performance of STORM with a set of micro-benchmarks. We focus our attention on two main aspects: job launching and gang scheduling. Finally, some concluding remarks and future directions are given in Section 4.

2. STORM Architecture

This section describes the architecture of STORM. The main design goals for STORM were the following:

1. Provide resource management mechanisms that are scalable, high-performance, and lightweight
2. Support the most of current and future job scheduling algorithms.

To fulfill the first goal, we use a set loosely coupled daemons that communicate with extremely fast messages. Coordination of the daemons is done through scalable strobes (heartbeat messages) that use hardware multicast. For the second goal, the daemons were designed so that modules for different scheduling algorithms can be "plugged" into them. In this paper, we focus on one of the most popular of these algorithms, gang-scheduling (GS)[7, 21]. GS employs both space sharing and time sharing to allocate resources to jobs.

All the processes of a given job run in the same allotted time slot for the duration of the timeslice quantum and are then context-switched to a different job in a cyclic manner at the end of each time slot.

2.1. Overview of STORM

Several issues were considered crucial for STORM, and were incorporated in its design and implementation:

1. **Flexibility:** The most important feature of STORM is the ability to support many modern and future scheduling algorithms in order to provide a valuable research tool. STORM currently supports local scheduling, First-Come-First-Served (FCFS or batch), FCFS with backfilling, GS, and Spin-Block (implicit coscheduling). Moreover, other scheduling methods can be readily added to the system. In fact, our research directions include the implementation of buffered coscheduling (BCS) [8, 9] and other scheduling algorithms.

2. **Scalability:** STORM is designed so that most of the scheduler operations are decentralized and asynchronous, and the only two global operations, namely job launching and strobes, are implemented by fast and lightweight hardware multicasts.

3. **Performance:** By using fast user-level communication and a low-overhead implementation, STORM is designed to be a lightweight, efficient scheduler.

4. **Simplicity:** The scheduler should not be over complicated, so that maintenance and augmentation of new scheduling algorithms will incur little overhead. This implies that parallel applications should not be changed to accommodate the system, and at most need only to be re linked.

5. **Portability:** The scheduler should be designed so that porting it to other hardware platforms, interconnects or even operating systems, will be relatively simple. To this end, STORM runs entirely in user level with no operating system modifications. Furthermore, the single hardware-dependent module of STORM, the underlying communication layer, is encapsulated in a small, isolated module. As of this writing, STORM runs on two hardware platforms (Intel x86 and Alpha EV6) and two networks (QsNET and a generic MPI layer).

2.2. The Quadrics Network

The QsNET is based on two building blocks, a programmable network interface called Elan [26] and a low-latency high-bandwidth communication switch called Elite [27]. Elites can be interconnected in a fat-tree topology [19].

The Elan network interface links the high-performance, multi stage Quadrics network to a processing node containing one or more processing elements (PEs). In addition to generating and accepting packets to and from the network, the Elan is equipped with a 32-bit thread processor, which is used to aid the implementation of higher-level messaging libraries without explicit intervention from the main CPU. The other building block of the QsNET is the Elite switch. The Elite provides the following features: (1) eight bidirectional links supporting two virtual channels in each direction, (2) a full crossbar switch, (3) a transmission bandwidth of 320 MB/s per link and a flow through latency of $35\ ns$, and (4) hardware support for collective communication.

Collective Communication Packets can be sent to multiple destinations using the *hardware* multicast capability of the network. A multicast packet can only take a pre determined path in order to avoid deadlocks. All nodes connected to the network are capable of receiving the multicast packet as long as the multicast set is physically contiguous. For a multicast packet to be successfully delivered, a positive acknowledgment must be received from all the recipients of the multicast group. The Elite switches combine the acknowledgments, as pioneered by the NYU Ultracomputer [4] [24], returning a single one to the source. Acknowledgments are combined in a way that the "worst" ACK wins (a network error wins over an unsuccessful transaction, which on its turn wins over a successful one), returning a positive ACK only when all the partners in the collective communication complete the distributed transaction with success. An in-depth experimental evaluation of the network [23] shows that the broadcast bandwidth scales almost linearly with the number of nodes, reaching an aggregate bandwidth that is linear with the number of destination nodes. The same infrastructure can also be used to notify the end of a timeslice in a gang scheduling or coscheduling algorithm.

2.3. Process Structure

STORM consists of three types of daemons that handle job launching, scheduling, and monitoring: the Machine Manager or MM (a single daemon on a management node), the Node Manager or NM (one daemon per compute node) and the Program Launcher or PL (several daemons per compute node).

The MM is in charge of resource allocation for jobs (both in space and time). Whenever a new job arrives, the MM queues it and tries to allocate PEs to it (using a buddy tree algorithm [10, 11]). If the scheduling policy allows for multiprogramming (e.g. GS), the PEs are allocated in any time slot that has enough available resources. After a successful allocation, the MM broadcasts a job-launch message to all the NMs, and those NMs on nodes that are allocated to

the job will launch it when its time slot arrives (the handling is done asynchronously). As an optimization, the MM can also broadcast binary image and data files to the relevant nodes before the execution of the program. Also, these nodes can cache the files on local disk or RAM disk, so that subsequent reruns can access the files locally. This optimization exploits the efficient hardware broadcast mechanism, instead of the non scalable use of NFS for distributing binaries. When a process of the job terminates, the MM receives an event from the corresponding NM, and frees the resources allocated to it. Note that even though the MM is centralized, in practice it does not create a bottleneck: all the global operations it performs are done with scalable hardware broadcasts and other operations such as reading a new job, allocating resources to it, and receiving process-termination notifications that are rare and lightweight.

NMs are responsible for managing resources on a single node (which could be an SMP). NMs work asynchronously, blocking until one of the following events arrive:

- Job launch: If the job pertains to the NM's node, the NM finds some available PLs and sends them the job information.
- Job caching: The binary image is read from the communication layer and stored in a file, preferably in a RAM-disk file to avoid unnecessary I/O.
- Heartbeat: The NM checks its local data structures, for every PE, whether a context switch is required. If so, it deschedules the current process (using UNIX's SIGSTOP) and resumes the next one.
- Process termination: upon receipt of such a message from the PL, the NM passes it on to the MM.

Some scheduling algorithms require that the NM makes its own local scheduling decisions. For example, in local scheduling, the NM ignores context-switch messages, as the local UNIX scheduler handles all scheduling decisions. In algorithms such as BCS, the NM might deschedule a process that is blocked for communication before the expiration of the time slot, and schedule another process instead.

The PLs have the relatively simple task of handling individual application processes for the NM. One copy of the PL runs for each PE and timeslot in a node, and sleeps until it receives a program execution event from the NM. It then proceeds to fork a new process, set up Quadrics communication capabilities for the application process (AP), redirect standard output and error to the console that launched STORM, and execute the AP. It then blocks with the wait system call until the AP terminates, notifying the NM when this happens.

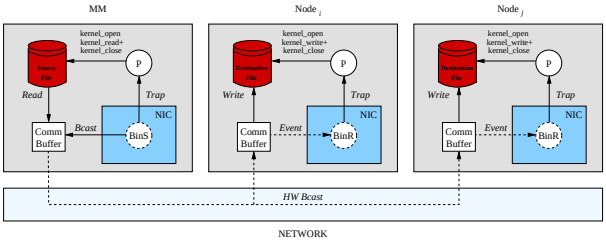

Figure 1. I/O bypass mechanism. kernel_read+ and kernel_write+ indicate sequences of kernel reads and writes. BinS and BinR are the Binary Sender and Receiver threads running on the Elan NIC.

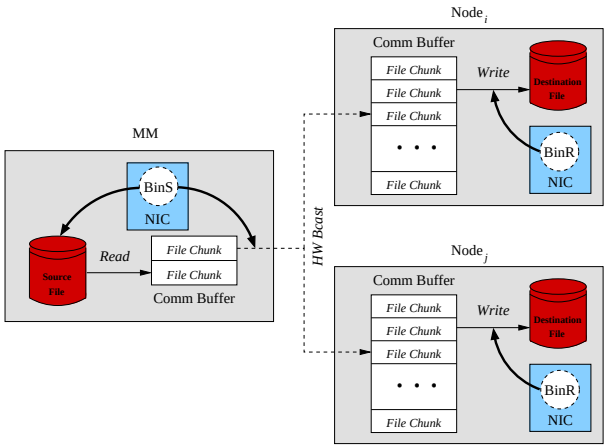

Figure 2. Pipelining of I/O read, hardware multicast and I/O write.

2.4. I/O bypass mechanism

We implemented a mechanism for alleviating one of the major bottlenecks in program launching, the interaction with the I/O subsystem. The threads in the Elan network interface can directly issue system calls that operate on the file system, for example, opening, reading, writing, and closing files. The relevant phases of the I/O bypass protocol during the launch of a job are listed below (see Figure 1).

1. The MM sends a DMA message to a thread in the local Elan NIC with the source file name and a remote destination path.
2. The sender thread uses kernel traps to open and read the source file. These traps go through the kernel, but require very little CPU intervention, so that the processes running on the processing node are not unaffected.

3. The file is read in chunks directly into a communication buffer that can be efficiently accessed by the Elan DMA engine, and then sent to a peer thread on all the compute nodes, using the hardware multicast.

4. The sender thread uses two chunk buffers to pipeline the reading and multicast operations, so that while one buffer is being read, the other is being sent in parallel, as shown in Figure 2.

5. The destination threads on the compute nodes queue the incoming chunks and write them to the destination path, using a flow-control protocol to avoid buffer overflows. File system writes and incoming multicasts can proceed in parallel.

6. When all the chunks have been sent and written to their respective local files, or conversely, if an error occurred, the MM is notified.

7. When the MM decides to launch the job (after successfully sending the binary and allocating resources to it), it uses the new remote path name in the job-launch message.

3. Experimental Results

In this section, we analyze the performance of STORM. In particular, we (1) measure the costs of launching jobs in STORM, and (2) test various aspects of the gang-scheduler (effect of the timeslice quantum, node scalability, and multiprogramming level).

3.1. Experimental Framework

The hardware used for the experimental evaluation was the 'crescendo' cluster at LANL/CCS-3. This cluster consists of 32 compute nodes (Dell 1550), one management node (Dell 2550), and the Quadrics network equipped with a 128-port switch [29] (using only 32 of the 128 ports). Each compute node has two 1.13 GHz Pentium-III with 1 GB of ECC RAM, two independent 66MHz/64-bit PCI buses, a Quadrics QM-400 Elan3 NIC [25, 26, 29] for data network and an Ethernet-100 network adapter for management network. All the nodes run under Red Hat Linux 7.1 with Quadrics kernel modifications and user-level libraries.

The application we use for the experiments in Sections 3.3-3.3 is SWEEP [16], a time-independent, Cartesian-grid, single-group, "discrete ordinates", deterministic, particle-transport code taken from the Department of Energy Accelerated Strategic Computing Initiative (ASCI) workload. SWEEP represents the core of a widely used method of solving the Boltzmann transport equation. Estimates are that deterministic particle transport accounts for $50-80\%$ of the execution time of many realistic simulations on current DOE systems.

Main Processor			Thread Processor		
NFS	Local	RAM-disk	NFS	Local	RAM-disk
11.22	30.50	506.00	11.43	31.5	120

Table 1. Read Bandwidth in MB/s for a $12\,MB$ Binary Image on NFS, Local-Disk, and RAM-Disk .

In the tests that involve a multiprogramming level (MPL) of more than one, we launch all the jobs concurrently (even though this may not be a realistic scenario), to further stress the scheduler.

3.2. Job Launching Time

In this set of experiments, we study the overhead associated with launching jobs with STORM and analyze its scalability with the size of the binary size and number of PEs. We use the approach taken by Brightwell et al. in their study of job launching on Cplant [6] by measuring the time it takes to run a program that terminates immediately, using different binary sizes: $4\,MB$, $8\,MB$, and $12\,MB$.

Anatomy of a job-launch The time taken for execution of a parallel job can be broken down into the following four components:

- *Read Time:* the time taken by the management node to read the binary from the file system. This image can be read through distributed file system like NFS, from a local disk, or it can be cached in a RAM disk.[5] Table 1 shows the read times of a $12\,MB$ binary on a compute node. We distinguish the two cases when a process or an Elan thread try to read the file in order to expose the performance of the I/O by-pass protocol. There is little difference between main and thread processors in the slow cases, namely NFS and the local disk. But processes can take advantage of the RAM disks, getting more than 500 MB/s, while the thread processor can only get 120 MB/s. We still have not fully investigated this asymmetry, which is influenced by the fact that the thread processor resides on the slower, PCI bus.

- *Broadcast Time*: the time to broadcast the binary image to all the compute nodes. This collective communication can take place in several ways, thus affecting how the time is measured. For example, if the file is read

[5]The RAM disk is a segment of RAM configured to simulate a UNIX file system. A RAM disk is expected to be faster than an actual disk drive.

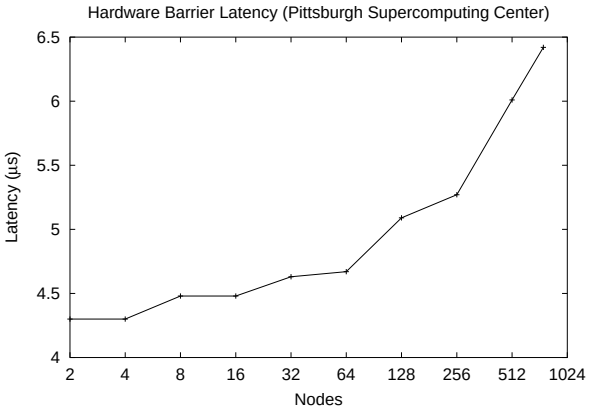

Figure 3. Barrier synchronization latency as a function of the number of nodes, Terascale computing system, Pittsburgh supercomputing center.

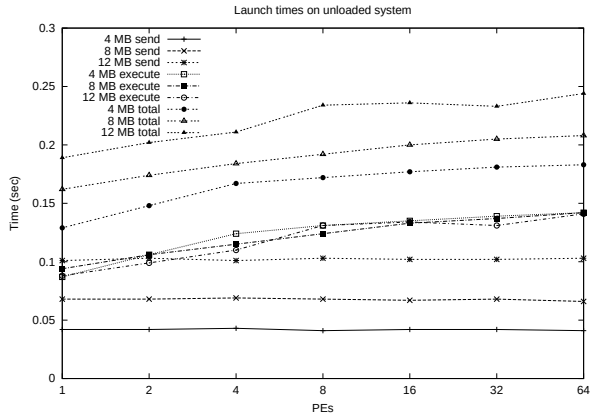

Figure 4. Send, execute and total launch times for 4MB, 8MB, and 12MB files

through a distributed file system like NFS, the distribution time and file read time are intermixed. However, if a dedicated mechanism is used to disseminate the file, like ParPar's [18] or our own, this component can be measured separately from the others. QsNET can broadcast messages in a scalable way and there is no significant performance penalty when increasing the number of nodes. The typical performance for a main-memory-to-main-memory broadcast is around 200 MB/s per node [23]. Figure 3 shows the scalability of the hardware multicast (Section 2.2), on the Terascale Computing System Installed at Pittsburgh Supercomputing Center (PSC), a cluster with 758 nodes. We can see that the latency grows by a negligible amount, about 2 μs. This is a reliable indicator that the broadcast, implemented with the same hardware mechanism, will scale efficiently.

- *Write Time*: write time is less critical than read time because the file copy on the client nodes is followed by an *exec* of the binary. Depending on how the kernel is implemented, part of the binary can reside on the buffer cache in memory at the time of the execution, and it does not need to be flushed to disk.

- *Execution overhead*: some of the time for launching a job in STORM is spent in allocating resources, waiting for a new time slot to launch it, and possibly for another time slot to run it. Events such as process termination are also collected by the MM at STORM timeslice intervals only, so a delay of up to 2 time-quanta is spent in MM overhead.

Our implementation tries to pipeline the three delays: read time, broadcast and write time, by dividing the file transmission into chunks of 128 KB. Table 1 shows that the bottleneck is the read time from disk in the management node, which is 118 MB/s vs 200 MB/s for the broadcast. Based on the scalability analysis reported in [23] and in Figure 3, we believe that this will be the bottleneck in large-scale (up to 4,096 nodes) configurations too.

Launch times in STORM As described in Section 2, STORM divides the job-launching task into two separate operations: the sending (broadcasting) of the binary image, which can be done before the designated time of running, and the actual execution, which includes sending the job-launch command, forking the job, waiting for its termination and reporting back to the MM. We measure the times of both these tasks on the MM, as well as their sum (representing the total time it takes to launch a job.) Figure 4 shows the time it takes to send each of the binaries, as well as the time to execute them and the total time to launch the job. Observe that the send times are roughly proportional to the binary size, but do not grow significantly with the number of nodes. This can be explained by the highly scalable hardware broadcast that is used for the send operation. On the other hand, the execution times are quite independent of the binary size, but grow slowly with the number of nodes. One reason for this growth is the cumulative time it takes for the MM to receive point-to-point notifications of the process termination from all the NMs. Another reason for the growth is the natural skew that occurs when jobs on different nodes take slightly different time to terminate (because of local UNIX considerations), thus statistically increasing the total run time as the number of nodes increases. Be-

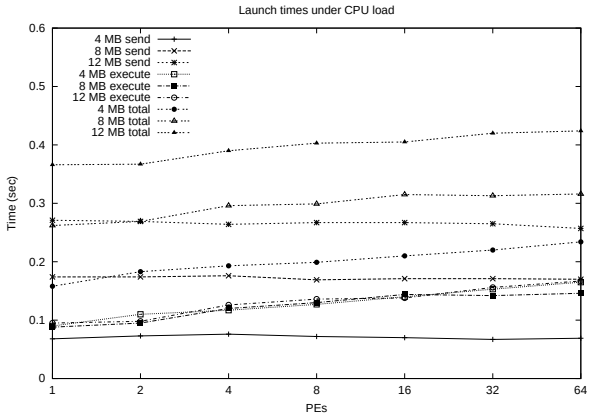

Figure 5. Send, execute and total launch times on a CPU-loaded system.

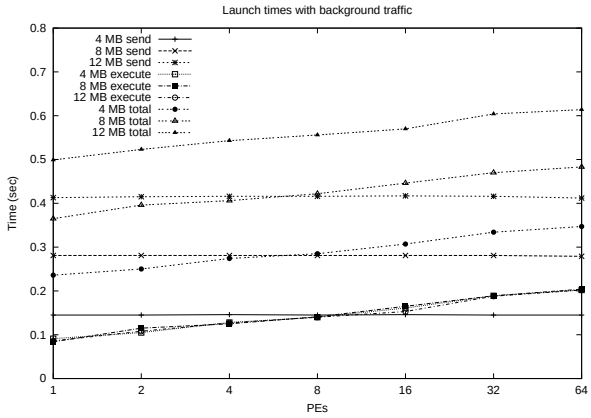

Figure 6. Send, execute and total launch times on a network-loaded system.

cause the increase is relatively small,[6] we did not address this issue for this version of STORM, but intend to replace the termination collection mechanism with a more scalable one, using Quadrics' hardware support for collective communication.

Launching on a loaded system To test how a heavily loaded system affects the launch times of jobs, we added two different programs that run in the background on all the cluster's nodes while measuring job-launch times. The first program performs a CPU-intensive computation.

Figure 5 shows the results of launching the same three binaries while the CPU-consuming program is running in the background. We can see that although the execution times remain nearly unaffected by the system's load, the send times are approximately doubled. This large increase is mostly due to the interference of the computation with the I/O activities (reads and writes). However, the total launch time for all programs is still quite small, and it is less than twice the launch time on an unloaded system.

The second program is designed to stress the entire network, by pairing all the processes and continuously sending long messages back and forth. This test is particularly interesting, because a previous study [23] shows that a heavily loaded network may have an adverse effect on collective communications in the Quadrics interconnect. In Figure 6 we can see how running this program in the background affects the launch time of the test binaries. Indeed, there is a small, but noticeable, increase in the execution times – due mostly to the increased delay in the delivery of termination messages from the NMs. However, the send operation

[6] Even with this exponential growth rate, it would still take less than 300 ms to launch a 12 MB binary on $4,096$ nodes.

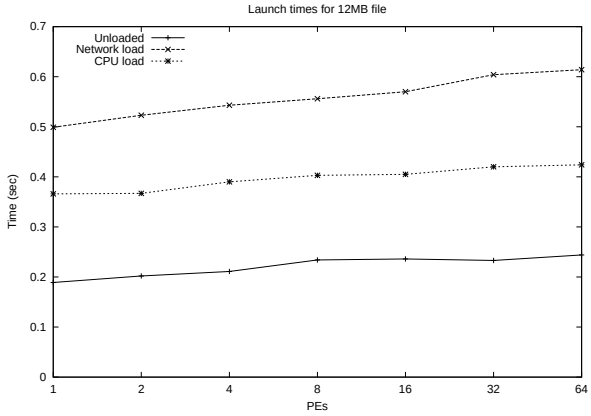

Figure 7. Total launch times for a 12 MB binary.

is considerably slower than on an unloaded system. This agrees with the previous study, because the communication part in the *send* operation is implemented by a Quadrics collective.

Figure 7 summarizes the difference between the launch times on loaded and unloaded systems. In this figure, the total launch time is shown for the 12-MB file only, under the three loading scenarios. Note that even in the worst scenario, with a network-loaded system, it still takes only ≈ 0.6 seconds to launch a 12-MB file on 64 nodes, and the growth rate of about 3.5% on every doubling of the nodes, suggest it would take less than a second to launch this program on a $4,096$-node machine (assuming the same growth rate).

3.3. Gang Scheduling Performance

Effect of Timeslice We first analyze the range of usable timeslice values, to better understand the limits of the gang-scheduler. For this experiment we use the SWEEP kernel, which uses extensive computation and communication resources. To test this, we modify the MM so that it does not sleep between heartbeat messages, but rather uses busy waiting. This requires that the MM have its own dedicated PE, so as not to interfere with the AP, but gives us the possibility of using timeslices smaller than the operating system resolution, in this case, one IA-32 Linux Jiffy, or $10\,ms$. Figure 8 shows the runtime for running SWEEP with the same input for various timeslices, up to 1.6 seconds, running on 32 PEs, with an MPL of 2. The smallest timeslice value that the scheduler can handle gracefully is $\approx 2\,ms$, below which the NM cannot process the incoming strobe messages at the rate they arrive. Note that this value is in the same order of magnitude of the local Linux scheduler's quantum or less, and approximately two orders of magnitude smaller than the smallest timeslice quantum reported in the literature [17, 14]. This allows for good system responsiveness, and usage of the parallel system for interactive jobs. Furthermore, a short quantum allows the implementation of advanced scheduling algorithms that can benefit greatly from short timeslice quanta, such as Buffered Coscheduling (BCS) [8, 9], Implicit Coscheduling (ICS) [1, 3], and Periodic Boost (PB) [20]. STORM can handle such small timeslice values due to the low-overhead processing of heartbeats in the MM and NMs, and the low-latency multicast mechanism provided by the Quadrics hardware.

From the graph we can see that starting from $\approx 10\,ms$, the observed run time grows as the timeslice grows. This is caused by the fact that events, such as process launch and termination reporting, only happen at timeslice intervals. On the other hand, using very small timeslices increases the overhead of context switching. The sweet spot between maximum responsiveness and minimum overhead is $\approx 8\,ms$.

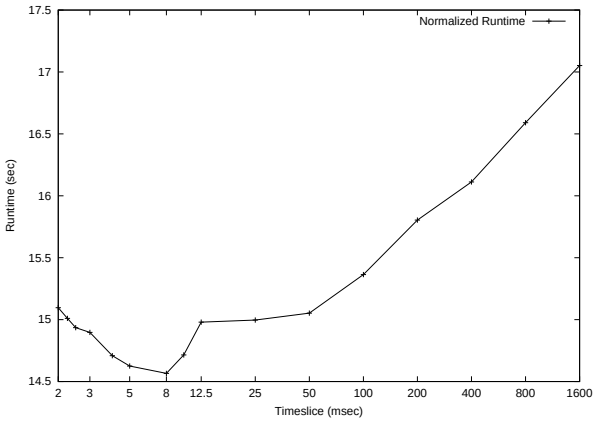

Figure 8. Effect of timeslice quantum with an MPL of 2, on 32 PEs.

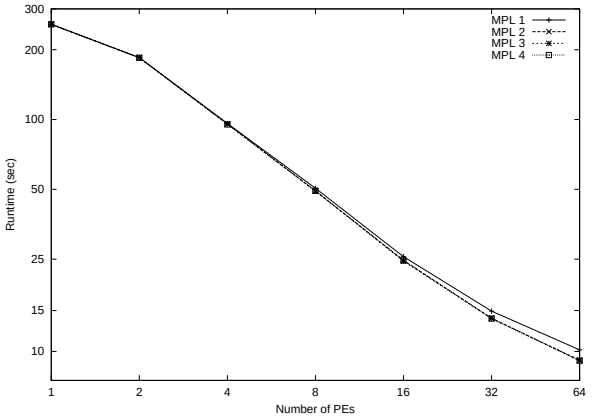

Figure 9. Effect of number of PEs on normalized runtime, for MPL values of 1-4.

Node Scalability and multiprogramming level An important metric of a resource manager is the scalability with the number of nodes and PEs it manages, and with the multiprogramming level. To test this, we measured the effect on the runtime of the program when running on an increasing number of nodes and under different MPLs. We use strong scaling in the experiment, i.e., SWEEP runs the same input on different numbers of nodes, and is thus expected to run faster as the number of nodes increases. Figure 9 shows the results for running the program on varying number of PEs in the range 4 to 64, for MPL values in the range 1 to 4 (results are normalized by dividing the runtime of jobs by the MPL, so that they are comparable on the same scale). The timeslice used for this experiment was $20\,ms$, providing a fairly responsive system.

Two interesting properties can be observed from these results. First, we note that increasing the MPL not only does not penalize the applications, but actually improves their normalized performance. This is due to the fact that the Quadrics communication library and hardware can overlap some of the communication and computation of different jobs [13, 14]. The context switch operation in STORM is rudimentary, requiring very little computation to determine the next process to run, suspending the current process and resuming the next one. This is actually less work than the UNIX scheduler typically takes for a context switch [28] so we may expect it to incur a small overhead. Thus, any overhead that might be caused by context-switching, including

working-set penalties, are even smaller than the gain from overlapping for an application like SWEEP. Another interesting property is that this gain actually grows as the number of PE increases. This is due to the fact that on the one hand, the increased number of nodes (and subsequent communication in SWEEP) allows the system to overlap more computation and communication. On the other hand, the strobe is implemented using a scalable hardware multicast operation and thus it does not significantly increase the overhead as the number of nodes grows. We can thus conclude that STORM scales very well both with the number of nodes and the multiprogramming level on the system tested, and contrary to intuition, using a multiprogramming level higher than one actually benefits communication-intensive applications like SWEEP.

4. Conclusions

In this paper we presented STORM, a flexible and scalable resource-management tool for large-scale clusters. With STORM we tried to prove the concept that it is possible to perform ultra fast resource management with latencies well under a second even in the presence of high CPU utilization or network congestion. The paper provided a number of technical guidelines on how to achieve these goals.

This paper also explored the performance of the gang-scheduling algorithm implemented in STORM. We showed that by using a set of loosely coupled daemons that exploit fast hardware communication mechanisms, we can implement an extremely efficient scheduler. This scheduler can handle timeslice quanta of less than 1 ms, providing responsiveness similar to that of the local UNIX scheduler. Furthermore, the low overhead associated with STORM tasks provides excellent scalability with the number of nodes.

We demonstrated that STORM encompasses major advances in resource management by using three novel techniques: (1) employing NIC threads to relieve the resource-manager from most communication tasks, (2) relying on efficient hardware collectives to perform global operations in a scalable way, and (3) using an I/O bypass mechanism that minimizes system overhead. The combination of these methods decreases the job-launch time by two orders of magnitude, thus making the system much more responsive and usable.

It is our hope that the flexibility inherent in the design of STORM will prepare the ground for new research results in the area of resource management and scheduling for large-scale parallel computers.

Future work Our main venues of research include the implementation and testing of new scheduling algorithms, in order to address critical resource-management issues such as reliability, load balancing, and system utilization. One important point in this direction is the implementation of user-transparent fault tolerance that seamlessly allows applications to resume execution even when nodes fail. Another direction is the implementation of a flexible coscheduling algorithm that can increase system utilization in the presence of load imbalance. We also plan to validate the scalability of STORM on larger clusters.

Acknowledgments We thank David Addison and David Hewson for their timely and insightful advice in the design of the I/O bypass protocol. We also thank Duncan Roweth for the barrier scalability analysis on the Terascale computing system Installed at PSC.

References

[1] Andrea C. Arpaci-Dusseau, David Culler, and Alan M. Mainwaring. Scheduling with Implicit Information in Distributed Systems. In *Proceedings of the 1998 ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems*, Madison, WI, June 1998.

[2] Andrea Carol Arpaci-Dusseau. Implicit Coscheduling: Coordinated Scheduling with Implicit Information in Distributed Systems. *ACM Transactions on Computer Systems (TOCS)*, 19(3), 2001.

[3] Remzi Arpaci-Dusseau, Andrea C. Arpaci-Dusseau, Amin Vahdat, Lok T. Liu, Thomas E. Anderson, and David A. Patterson. The Interaction of Parallel and Sequential Workloads on a Network of Workstations. In *Proceedings of the 1995 ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems*, pages 267–278, Ottawa, Canada, May 1995.

[4] G. Bell. Ultracomputer: a Teraflop before its time. *Communications of the ACM*, 35(8):27–47, 1992.

[5] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawick, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, January 1995.

[6] Ron Brightwell and Lee Ann Fisk. Scalable Parallel Application Launch on Cplant. In *Supercomputing 2001*, Denver, CO, November 2001.

[7] Dror G. Feitelson and Larry Rudolph. Gang Scheduling Performance Benefits for Fine-Grain Synchronization. *Journal of Parallel and Distributed Computing*, 16(4), 1992.

[8] Fabrizio Petrini and Wu-chun Feng. Buffered Coscheduling: A New Methodology for Multitasking Parallel Jobs on Distributed Systems. In *Proceedings of the International Parallel and Distributed Processing Symposium 2000, IPDPS2000*, volume 16, Cancun, MX, May 2000.

[9] Fabrizio Petrini and Wu-chun Feng. Improved Resource Utilization with Buffered Coscheduling. *Journal of Parallel Algorithms and Applications*, 2000.

[10] Dror G. Feitelson. Packing Schemes for Gang Scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing – Proceedings of the IPPS'96 Workshop*, volume 1162, pages 89–110. Springer, 1996.

[11] Dror G. Feitelson, Anat Batat, Gabriel Benhanokh, David Er-El, Yoav Etsion, Avi Kavas, Tomer Klainer, Uri Lublin, and Marc Volovic. The ParPar System: a Software MPP. In Rajkumar Buyya, editor, *High Performance Cluster Computing*, volume 1: Architectures and systems, pages 754–770. Prentice-Hall, 1999.

[12] Dror G. Feitelson and Morris A. Jette. Improved Utilization and Responsiveness with Gang Scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pages 238–261. Springer-Verlag, 1997.

[13] Eitan Frachtenberg and Fabrizio Petrini. Overlapping of Computation and Communication in the Quadrics Network. Technical Report LAUR 01-4695, Los Alamos National Laboratory, August 2001.

[14] Eitan Frachtenberg, Fabrizio Petrini, Salvador Coll, and Wu chun Feng. Gang Scheduling with Lightweight User-Level Communication. In *2001 International Conference on Parallel Processing (ICPP2001), Workshop on Scheduling and Resource Management for Cluster Computing*, Valencia, Spain, September 2001.

[15] Douglas P. Ghormley, David Petrou, Steven H. Rodrigues, and Amin M. Vadhar. GLUnix: a GLobal Layer Unix for a Network of Workstations. *Software - Practice and Experience*, 28(9), 1998.

[16] Adolfy Hoisie, Olaf Lubeck, and Harvey Wasserman. Scalability Analysis of Multidimensional Wavefront Algorithms on Large-Scale SMP Clusters. In *The Ninth Symposium on the Frontiers of Massively Parallel Computation (Frontiers'99)*, Annapolis, MD, February 1999.

[17] Atsushi Hori, Hiroshi Tezuka, and Yutaka Ishikawa. Highly Efficient Gang Scheduling Implementation. In *Supercomputing 98*, Orlando, FL, November 1998.

[18] Avi Kavas, David Er-El, and Dror G. Feitelson. Using Multicast to Pre-Load Jobs on the ParPar Cluster. *Parallel Computing*, 27:315–327, 2001.

[19] Charles E. Leiserson. Fat-Trees: Universal Networks for Hardware Efficient Supercomputing. *IEEE Transactions on Computers*, C-34(10):892–901, October 1985.

[20] Shailabh Nagar, Ajit Banerjee, Anand Sivasubramaniam, and Chita R. Das. A Closer Look At Coscheduling Approaches for a Network of Workstations. In *Eleventh ACM Symposium on Parallel Algorithms and Architectures, SPAA'99*, Saint-Malo, France, June 1999.

[21] J. K. Ousterhout. Scheduling Techniques for Concurrent Systems. *Proceedings of Third International Conference on Distributed Computing Systems*, pages 22–30, 1982.

[22] Fabrizio Petrini, Wu chun Feng, Adolfy Hoisie, Salvador Coll, and Eitan Frachtenberg. The Quadrics Network: High Performance Clustering Technology. *IEEE Micro*, 22(1):46–57, January-February 2002.

[23] Fabrizio petrini, Salvador Coll, Eitan Frachtenberg, and Adolfy Hoisie. Hardware-Based and Software-Based Collective Communication on the Quadrics Network. In *Proceedings of the IEEE International Symposium on Network Computing and Applications*, Cambridge, MA, October 2001.

[24] G. F. Pfister and V. A. Norton. Hot-spot Contention and Combining in Multistage Interconnection Networks. *IEEE Transactions on Computers*, C-34(10):943–948, October 1985.

[25] Quadrics Supercomputers World Ltd. *Elan Programming Manual*, January 1999.

[26] Quadrics Supercomputers World Ltd. *Elan Reference Manual*, January 1999.

[27] Quadrics Supercomputers World Ltd. *Elite Reference Manual*, November 1999.

[28] Jeffrey H. Straathof, Ashok K. Thareja, and Ashok K. Agrawala. UNIX Scheduling for Large Systems. In *Proceedings of the USENIX Winter Conference*, pages 111–139, Denver, CO, 1986.

[29] http://www.quadrics.com.